UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

FUNCTIONAL DOMAIN DRIVEN DESIGN

Sergio Herrera Guzmán
Tutor: Juan José Vázquez Delgado
Ponente: Francisco Saiz López

Junio 2016

FUNCTIONAL DOMAIN DRIVE DESIGN

AUTOR: Sergio Herrera Guzmán
TUTOR: Juan José Vázquez Delgado
PONENTE: Francisco Saiz López

Empresa Tecnología, Sistemas y Aplicaciones S.L.
Dpto. de desarrollo de software
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2016

Resumen

Las tecnologías están en constante expansión y evolución, diseñando nuevas técnicas para cumplir con su fin. En el desarrollo de software, las herramientas y pautas para la elaboración de productos software constituyen una pieza en constante evolución, necesarias para la toma de decisiones sobre los proyectos a realizar.

Uno de los arquetipos para el desarrollo de software es el denominado Domain Driven Design, donde es importante conocer ampliamente el negocio que se desea modelar en forma de dominio, a través de ciertos elementos como la identificación de entidades o un lenguaje cuidado.

La gran evolución de la computación en la nube y los servicios con funciones específicas, denominados microservicios, introducen ciertas necesidades en los algoritmos computacionales que manejan, como la seguridad en el procesamiento de los datos. Para alcanzar estos requisitos evitando mecanismos de control costosos, la programación funcional provee de múltiples elementos que confieren al lenguaje de capacidades para alcanzar este fin, como inmutabilidad de los datos.

La unión de estos elementos y requisitos produce lo que se conoce como Functional Domain Driven Design. Una aplicación de este modelo de desarrollo de software será Command-Query Responsability Segregation, un patrón basado en la separación del dominio en dos subsistemas diferenciados, un sistema responsable del control del flujo de ejecución y otro centrado en las consultas a realizar sobre el primero.

La computación distribuida y los microservicios también necesitan escalar dado el creciente número de usuarios del sistema. En ocasiones, escalar verticalmente no está ligado a aumentar el rendimiento y, por ello, la escalabilidad horizontal es un punto fuerte para estas unidades computacionales, al poder trabajar conjuntamente, replicándose, para cumplir su función entre múltiples sistemas, balanceando la carga de los mismos. Distributed Domain Driven Design entra en juego para proveer de técnicas para la implementación del mismo.

Palabras clave

Desarrollo, software, Domain Driven Design, negocio, modelo, computación en la nube, programación funcional, Command-Query Responsability Segregation, computación distribuida, escalar, Distributed Domain Driven Design

Abstract

Technologies are in constant expansion and evolution, designing new techniques to comply with their aim. In software development, the tools and the guidelines for creating software products establish a piece which is in constant evolution, needed to make decisions about the projects to be created.

One of the archetypes for software development is known as Domain Driven Design, where a big knowledge about the business to de modelled is needed in form of domain, with the knowledge of some elements like the identification of entities or a precise language.

Great evolution in cloud computing and services with specific roles, denominated microservices, import certain concerns in the computing algorithms in which they operate, like the security in data processing. To achieve these requirements avoiding expensive control mechanisms, functional programming brings several elements, like immutable data, that grant languages facilities to reach this aim.

The union of these elements and requirements produces what is called Functional Domain Driven Design. A variant that follows this software development model is Command-Query Responsability Segregation, a pattern based on the separation of the Domain into two distinct subsystems, one system responsible of controlling the execution flow, and another centered in query things about the first.

The distributed computation and microservices also need to be scaled due to the growing number of system users. Scaling vertically is sometimes not linked to increase the performance and, due to this, scaling horizontally is a particular strength for these computational units, working together, replicating, to fulfil its function among multiple systems, balancing the load of the same. Distributed Domain Driven Design comes into play to provide the technics for implementing it.

Keywords

Development, software, Domain Driven Design, business, model, cloud computing, functional programming, Command-Query Responsability Segregation, distributed computing, scale, Distributed Domain Driven Design

Agradecimientos

En primer lugar, agradecer en especial a mi tutor Juan José Vázquez por sus grandes ideas, conocimiento y ofrecerme la posibilidad de realizar este trabajo con él en Tecsisa.

A mis compañeros de estudios y trabajo en Tecsisa, que siempre me han ayudado y apoyado demostrándome de lo que soy capaz, en especial a Raúl Hernando, Mario Polo, Francisco José Bermejo e Iván Muñoz.

Al profesor de la EPS-UAM, Francisco Saiz, cuya labor y esfuerzo influyó en la decisión de adquirir conocimientos sobre la programación funcional.

A mi familia, que desde pequeño me han dado cariño y apoyo en todas las decisiones que he tomado.

Sergio Herrera Guzmán

Junio 2016

ÍNDICE

ÍNDICE DE FRAGMENTOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN

1.1. Motivación

Este trabajo se enmarca en el área del análisis, diseño y desarrollo de software como producto final y en los marcos de la Ingeniería de Software (IS) y la Computación (CO). En el desarrollo de software es una evolución constante de técnicas y metodologías para la elaboración del mismo. *Domain Driven Design* (*DDD*) comprende una serie de técnicas, metodologías y patrones a través de los cuales el desarrollo de productos software debe ser una tarea sencilla si se conoce el campo con el que se trata. Este modelo de desarrollo de software encaja en el ámbito de la programación orientada a objetos (en inglés, *Object Oriented Programming*, *OOP*) pero en base al auge de las técnicas de programación funcional (en inglés, *Functional Programming*, *FP*), dada la modularidad y seguridad de los algoritmos, y las arquitecturas concurrentes, implicando infraestructuras de múltiples máquinas, múltiples núcleos o combinaciones de ambas, es necesario analizar y destacar la necesidad de evolucionar este modelo a otras perspectivas.

Estas necesidades de diseño se encuentran influenciadas por el auge de software basado en microservicios en la industria del desarrollo, unidades computacionales independientes con un ámbito muy limitado al conocimiento del negocio que trata. Esto implica que un microservicio encaja con una funcionalidad muy específica y debe poseer algoritmos capaces de ejecutarse múltiples veces con seguridad de obtener resultados fiables y esperados. En estos casos, aplicar *DDD* mediante patrones funcionales puede significar un acierto como metodología de desarrollo de la unidad computacional.

Dado el pequeño tamaño de los microservicios, es necesario de disponer de múltiples instancias de cada unidad funcionando de forma coordinada, consiguiendo un modelo distribuido de software. Debido a ello es necesario introducir lo denominado *Distributed Domain Driven Design* (*DDDD*). Dada la complejidad de este, es importante explicar la importancia del Teorema *CAP* y su influencia en esta técnica.

1.2. Objetivos

Con este trabajo se pretende formalizar una serie de pautas, técnicas y patrones que servirán de base para diseñar una librería de desarrollo de software siguiendo las indicaciones de *DDD* a través de lenguajes funcionales. Para alcanzar este fin, se extraerán los fundamentos de *DDD* y se analizará cuidadosamente su encaje en la programación funcional.

Para el desarrollo de esta librería se empleará un lenguaje con características para la aplicación de técnicas funcionales, *Scala*, y un conjunto de herramientas que serán de ayuda para la programación concurrente y distribuida, *Akka* y *Cats*. También tiene cabida la explicación de algunos elementos externos que actuarán para persistir los datos generados, principalmente, bases de datos en memoria y la base de datos no relacional *Apache Cassandra*.

Finalmente, se asentarán las bases para el trabajo futuro, de forma teórica, a través de la investigación de *DDDD* y cómo la implementación del teorema *CAP* puede afectar en función de los requisitos que se desean introducir en las aplicaciones.

1.3. Estructura del Trabajo

Este trabajo está organizado en seis Capítulos. A continuación, se describe el contenido de cada uno de ellos.

El Capítulo 1 incluye una presentación del problema a tratar, los objetivos de este trabajo y una breve estructura de cómo se organiza este documento.

El Capítulo 2 aborda el estado de la cuestión. Se introducen las definiciones y términos necesarios para entender *DDD*, la programación funcional y las tecnologías implicadas en el desarrollo de este trabajo.

En el Capítulo 3 se analiza cómo puede encajar este modelo de desarrollo de software con las técnicas de programación funcional, involucrando un lenguaje con características de programación funcional y múltiples librerías en las que se apoya para alcanzar el objetivo.

El Capítulo 4 detalla un diseño basado en los elementos propuestos en el capítulo anterior para introducir patrones necesarios y técnicas que nos sirven de base para emplear este modelo de desarrollo de software.

El Capítulo 5 contiene una serie de elementos adicionales a través de los cuales se define *DDDD*, la aplicación del modelo expuesto en entornos distribuidos basados en microservicios.

El Capítulo 6 presenta las conclusiones recogidas durante la realización de este trabajo y tareas futuras a desarrollar.

CAPÍTULO 2. ESTADO DE LA CUESTIÓN

2.1. Domain driven design

El desarrollo dirigido por dominios, en inglés, *Domain Driven Design* (DDD), comprende una serie de técnicas para el desarrollo de software con necesidades complejas, basado en el modelado de dominios. Esta definición da pie a múltiples dudas como qué es un dominio o qué elementos lo componen. A continuación, se muestran varios ejemplos para intentar formalizar una definición de estos términos.

2.1.1 Qué es un dominio

Al desarrollar una aplicación para la venta de productos se debe tener en cuenta los conceptos del día a día de cualquier persona al ir a realizar la compra a cualquier tienda: coger un carro de la compra, añadir productos en el carro, modificar la cantidad de productos que contiene el carro, retirar productos del carro e incluso finalizar la compra con los productos que contiene el carro. Todas estas acciones anteriores pueden ser modeladas bajo un dominio, el dominio de la compra.

Ahora se propone otra situación de la vida cotidiana de una persona, el trato con un banco y las cuentas bancarias. Cuando una persona abre una cuenta bancaria, éste puede añadir dinero a la misma, extraer dinero de un cajero, realizar trasferencias a otras cuentas, entre otras operaciones. Todas ellas pertenecen a otro dominio completamente diferenciado del anterior, este es el dominio de la cuenta bancaria.

Tomando como base las ideas explicadas en los ejemplos anteriores se puede indicar que un dominio es un modelo de negocio complejo de la vida real el cual es posible expresar computacionalmente.

Al disponer de una definición de dominio basada en ejemplos, se van a identificar los elementos que conforman un dominio a través de los dos ejemplos mencionados anteriormente.

2.1.2. Elementos del dominio

Un dominio está comprendido por múltiples elementos fácilmente identificables:

- Objetos que se mantendrán a lo largo del dominio. En el dominio de la compra surgen los carros de la compra o los productos de la tienda. En el dominio del banco, aparecen los conceptos de cuenta, cantidad de dinero o persona.

- Comportamientos de los objetos a lo largo del dominio, esto serán las interacciones que pueden darse con los objetos anteriores, sea uno o varios. En el dominio de la compra se dispone de la interacción de coger un carro o añadir un producto al carro. En el dominio del banco se puede depositar dinero en una cuenta o transferir dinero a otra cuenta, sea propia o ajena.
- Un lenguaje que el dominio entiende, ajustado a las necesidades del mismo. Por ejemplo, para el dominio de la compra existen términos como 'carro', 'producto', 'añadir' o 'retirar' mientras que en el de las cuentas bancarias tenemos 'cuenta', 'dinero' o 'depositar'.
- Contexto en el que el dominio opera, es decir, una serie de supuestos y restricciones relevantes para el dominio con el que estamos trabajando. Una restricción para el dominio de la compra es la cantidad de productos disponibles. En el dominio del banco la capacidad de crear una cuenta a una persona o entidad y no a un animal es un supuesto a tener en cuenta.

En base a la definición de dominio y los elementos que componen un dominio, el siguiente apartado expresa la concepción de *DDD* y la relación de los componentes del dominio con los de *DDD*.

2.1.3. Definición y componentes de DDD

Al desarrollar software basado en modelos se debe tener cierto grado de habilidad para entender cómo funciona un modelo en la vida real. Conocer el funcionamiento del dominio y ser capaz de abstraer las características principales del mismo en un modelo es lo que se conoce como *DDD* [1].

Para trabajar con *DDD* es necesario definir un conjunto de conceptos que conforman un dominio [2] dentro del marco de este modelo de desarrollo de software:

- *Bounded Context*, de forma individualizada, es un módulo que forma el sistema y define al más alto nivel de granularidad una funcionalidad completa.
- Elementos del domino, esto son los objetos que el dominio utiliza para trabajar. Entre ellos hay que destacar tres elementos principales.
    - Entidad, elemento que se caracteriza por ser identificable por un único valor o un conjunto de estos. A partir del ejemplo de la compra, un carro de la compra es identificable por el usuario del carro, mientras que, en el banco, en cada cuenta tiene asociado un número como identificador único.
    - Objeto de valor, elemento similar a la entidad, salvo que es identificable por el conjunto completo de su contenido y, en caso de modificar cualquier valor del mismo, se obtendría otro elemento distinto. En el ejemplo de la compra, un producto es un objeto de valor dado que al cambiar alguna de sus partes se obtendría un producto nuevo. En el banco se puede determinar que una dirección es única por el conjunto de todos los datos.
    - Servicio, elemento de más alto nivel que, normalmente, modela un caso de uso del negocio en cuestión, y trabaja conjuntamente con entidades y objetos de valor. Por ejemplo, el servicio de gestión del carro de la compra, donde es posible iniciar una compra, añadir y eliminar productos, o finalizar la compra. En el caso del banco, con el servicio de gestión de cuentas el cliente puede

abrir una cuenta, ingresar o extraer dinero, realizar transferencias a otras y muchas más acciones.

- Elementos de gestión del ciclo de vida, mediante los que se puede mantener la duración de entidades y de objetos de valor en el sistema a través de múltiples patrones. De ellos destacan:
    - Factorías, las cuales permiten mantener en una misma localización la creación de entidades similares, siendo este un servicio de creación y, si es posible, inicialización de instancias. Por ejemplo, una factoría para la creación carritos o para crear los distintos tipos de cuentas bancarias, cuentas corrientes, cuentas de ahorro, etc.
    - Agregados, que permiten definir y acotar consistentemente la lógica que las entidades u objetos de valor pueden tomar. De esta forma un agregado puede manejar la consistencia sobre la cantidad de elementos disponibles para su compra o controlar las fechas de creación y cierre de una cuenta bancaria. Entre los agregados existe un tipo especial que se conoce como Agregado Raíz, el cual se encarga de actuar como fachada para que el usuario acceda al agregado correspondiente y además de controlar la mayor parte de reglas de negocio.
    - Repositorios, que se emplean para persistir y conservar entidades, objetos de valor o agregados cuando estos ya no son necesarios para el estado del negocio, pero si puedan serlo en un estado futuro. El tipo de repositorio más utilizado son bases de datos, sean relacionales o no relacionales, o algún elemento en los sistemas de ficheros. Al definir un repositorio es una buena práctica definir una fachada por si fuese necesario modificar el destino de los datos.
- Lenguaje ubicuo, con el que se puede definir un lenguaje adecuado y acotado al negocio a abstraer en el sistema. Siguiendo con los ejemplos anteriores, en el primer caso disponemos de palabras como *carro*, *producto*, *iniciar*, *añadir*, *eliminar* o *finalizar*, mientras que en el segundo aparecen algunas como *cuenta*, *crear cuenta*, *ingresar dinero*, *extraer dinero*, *transferir dinero* o *cancelar cuenta*. Con este lenguaje se intenta definir interfaces expresivas de modo que un programador pueda entender el negocio sin necesidad de conocer la implementación de la *API*, lo que se conoce como un *DSL*.

2.2. Programación funcional

La programación funcional es un paradigma de programación declarativo basado en el tratamiento de la computación como si fuesen funciones matemáticas, evitando modificar el estado de los datos en el interior de la función, lo que se conoce como transparencia referencial [3].

La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en el año 1936 por Alonzo Church, pensado para investigar la definición de función, su aplicación y la recursión.

En los siguientes apartados se describen diferentes conceptos relacionados con la programación funcional.

2.2.1. Funciones puras

El concepto de función pura expresa que una función no tiene efectos de lado, es decir, dado un argumento *A* introducido en una función *f*, esta produce un resultado *B* y, siempre que se aplique dicha función *f* con la misma entrada *A*, esta producirá como salida el mismo *B*, independiente del estado del sistema. Este hecho tiene aplicaciones útiles como la posibilidad de ejecutar evaluaciones de múltiples funciones si no existe dependencia entre ellas, lo que indica que dichas funciones son *thread-safe*, esto es, pueden ejecutarse de forma concurrente sin interferir una en la computación de la otras. También supone la posibilidad de realizar pruebas unitarias de las funciones de una forma más segura y fiable ya que no pueden producir estos efectos imprevistos.

2.2.2. Funciones de orden superior

Las funciones de orden superior o funtores son aquellas que, como mínimo, permiten como entrada una función o la retornan como salida [4]. Esto se expresa tal que:

- Si recibe una función de entrada, como en la función *map*, que a partir de un conjunto de elementos *[1, 2, 3]* y una función *f* capaz de tratar estos elementos, su resultado sería una nueva lista con la función aplicada sobre cada elemento, *[f(1), f(2), f(3)]*, esta se define como
  `map: A ⇒ (A ⇒ B) ⇒ B.`
- Si produce como salida una función, por ejemplo, la función *curry*, que separa la lista de argumentos de una función, se define como
  `curry: ((A, B) ⇒ C) ⇒ A ⇒ B ⇒ C.`

A los lenguajes que permiten estos hechos y, además, asignar funciones a variables, se les dice que tratan a las funciones como ciudadanos de primera clase. Este término fue dado por el informático teórico Christopher Strachey en la década de 1960.

2.2.3. Recursión

La recursión es el modo en que los lenguajes funcionales permiten iterar sobre el mismo bloque de código, simulando los bucles de la programación imperativa. De este modo, si un problema puede resolverse mediante recursión, este puede descomponerse en pequeñas instancias del mismo problema.

En la Tabla 1 se muestran dos ejemplos de la función factorial, de la que en primer lugar se dispone tanto de una implementación a través de bucles como de una solución recursiva, tras descomponer el problema en pequeños fragmentos.

Tabla 1. Implementación de factorial con bucles (izquierda) y mediante recursividad (derecha)

```
// Implementación con bucles
def factorial(x: Int) = {
  var n = x
  var total = 1
  while(n >= 1) {
    total *= n
    n -= 1
  }
  total
}
```

```
// Implementación recursiva
def factorial(x: Int): Int = {
  if(x <= 1) 1
  else x * factorial(x - 1)
}
```

Un problema que supone la recursión es que esta necesita mantener las llamadas sucesivas en la pila, lo que puede provocar un desbordamiento de la misma. Para solventar este problema, los compiladores pueden implementar y optimizar la recursión de cola, donde uno de los elementos de la función recursiva actúa como acumulador, sustituyendo las sucesivas llamadas en la misma posición de la pila. No todos los algoritmos recursivos pueden ser resueltos con recursión de cola, por ello los compiladores pueden poseer algún mecanismo de control que ayuda al programador indicándole si la función está implementada en esta forma. El Fragmento 1 muestra el ejemplo anterior en su variante recursiva con la implementación de la recursión de cola.

Fragmento 1. Implementación del factorial con recursión de cola

```
import scala.annotation.tailrec

@tailrec def factorial(x: Int, acc: Int): Int = {
  if(x <= 1) acc
  else factorial(x - 1, acc * x)
}
```

En *Scala*, a través de la anotación *@tailrec* el compilador produce un error al compilar si el algoritmo implementado no posee recursión de cola.

2.2.4. Sistema de tipos

Los lenguajes funcionales tienden a utilizar un sistema de tipos basado en el cálculo lambda tipado, esto hace que el propio compilador sea capaz de inferir los tipos ligados a la función y evitar el uso indebido de la misma.

El Fragmento 2. Ejemplo de inferencia del tipo de devolución muestra la definición de la función suma de dos números.

Fragmento 2. Ejemplo de inferencia del tipo de devolución

```
val sum =
  (x: Int, y: Int) ⇒
    x + y
sum: (Int, Int) ⇒ Int
```

Dadas las restricciones del lenguaje *Scala*, solo se especifican los tipos de entrada y en base a estos, podemos inferir el tipo de salida. La definición de la función anterior sería la siguiente:

En caso de definir la entrada con algún tipo más restrictivo, es decir, un tipo numérico de precisión doble, el propio sistema de inferencia de tipos se ajusta a este: `sum: (Int, Double) ⇒ Double`

2.2.5. Transparencia referencial

La transparencia referencial es la imposibilidad de reasignar valores a una variable. Los lenguajes funcionales no permiten la modificación del valor de una variable una vez este sea definido, lo que imposibilita que sea alterada en el cuerpo de una función, eliminando parte de efectos laterales.

Fragmento 3. Ejemplo de trasparencia referencial directamente sobre valores

```
val x = 3
val y = x + 2 // Creación de nuevo valor
x = 1 // ERROR: Reasignación de valores
```

En el Fragmento 3, se inicializan dos símbolos, *x* con valor *3* e *y* a partir del valor de *x* sumado *2*. De esta forma, *y* es un nuevo símbolo que no modifica el valor de *x*. La siguiente línea demuestra que al intentar asignar el valor *1* al símbolo *x* el compilador advierte de una reasignación de valores no permitida.

2.3. Tecnologías

En este apartado se evalúan y detallan las distintas tecnologías que están involucradas en el desarrollo de este trabajo.

2.3.1. Scala

Scala es un lenguaje de programación multiparadigma publicado en el año 2004 y diseñado por Martin Odersky, quien, trabajando en el desarrollo de funcionalidades muy utilizadas de *Java*, como los tipos genéricos, se dio cuenta de múltiples problemas que podía sufrir el lenguaje y decidió crear uno nuevo. En sus orígenes, *Scala* funcionaba sobre la *Java Virtual Machine* y sobre la *.NET platform*, abandonando el soporte a esta última en el año 2012. Su nombre proviene del acrónimo *"Scalable Language"* [5].

Figura 1. Logotipo de *Scala*

Algunas características destacables de este lenguaje son:

- Sistema de tipos estático muy fuerte, inferencia de tipos.
- Capacidades para programación orientada a objetos, orientada a actores y funcional.
- Soporte para programación funcional a través de inmutabilidad, currificación de funciones, evaluación perezosa, tipos de datos algebraicos, funciones de orden superior, constructores de tipos, *for-comprehension*.
- Otras características principales en las funciones son los parámetros opcionales o llamadas a funciones con parámetros por nombre, funciones anónimas, paso de funciones en llamadas a funciones.
- Notación infija, sobrecarga de operadores, herencia y polimorfismo.

En *Scala* todo es una expresión que retorna un valor, en caso de no existir un valor que retornar, el tipo *Unit* es un tipo *singleton* con un valor vacío.

Scala posee una librería nativa para la concurrencia a través del paquete *scala.concurrent*, en el que se incluyen *Future* y *Promise*. Con estos elementos se pueden realizar acciones asíncronas, lo que permite comunicaciones no bloqueantes con elementos del mundo exterior como los *DBMS*.

2.3.2. Akka

Akka es un conjunto de herramientas pensadas para la programación concurrente y distribuida, basado en actores. Jonas Boner, inspirado por el sistema de actores de *Erlang*, quiso portar su funcionalidad para proveer a *Scala* y *Java* de aplicaciones capaces de exprimir al máximo los recursos de la *JVM*. Su nacimiento se produjo en el año 2009 [6].

Figura 2. Logotipo de *Akka*

El sistema de actores de *Akka* permite la concurrencia a través de paso de mensajes entre actores y una serie facilidades para trabajar con una *JVM* local o un clúster constituido por múltiples *JVM*. Para ello, *Akka* aporta múltiples librerías modulares dependiendo de las necesidades del programador. Las principales librerías que serán de aplicación en este proyecto son las siguientes:

- *Akka actors* aporta la funcionalidad para desarrollar actores, así como el núcleo para trabajar con el resto de librerías, por ejemplo, el *ActorSystem*.
- *Akka persistence* ofrece una interfaz para trabajar con actores persistentes, *PersistentActor*, los cuales se alimentan de la recepción de comandos y persistiendo eventos. También permiten recobrar el estado del actor a partir de los eventos que ha persistido o de imágenes *snapshots* que puede genera.
- *Akka remote* incluye toda la funcionalidad base para trabajar con sistemas de actores en entornos distribuidos.

- *Akka cluster* es una especialización de la librería anterior que permite la clusterización sencilla de algoritmos, permitiendo su ejecución en entornos distribuidos.
- *Akka streams* ofrece una librería de streaming, implementación de los *Reactive Streams*, un estándar para el procesamiento asíncrono en *streaming* con técnicas de *backpressure* no bloqueante.

2.3.3. Cats

Cats es una librería que provee de abstracciones para ayudar a la programación funcional en *Scala*, según la teoría de categorías. Desarrollada principalmente por la comunidad *Typelevel* e iniciada en enero de 2016 por Erik Osheim, Cody Allen y Adelbert Chang. Su nombre procede de un acrónimo de la palabra categoría [7].

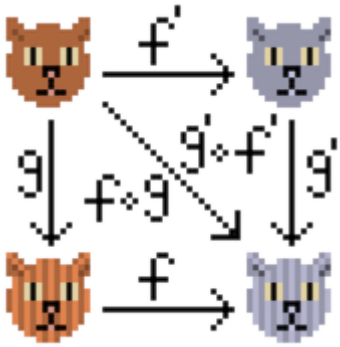

Figura 3. Logotipo de *Cats*

La librería está compuesta por múltiples módulos con funcionalidades específicas, como la definición de *typeclasses*, el conjunto de leyes para testear estas o aquel que contiene específicamente las estructuras *Free*, como la *Free Monad*.

2.3.4. Apache Cassandra

Cassandra es una base de datos *open source*, no relacional, distribuida, basada en el modelo de almacenamiento clave-valor y orientación a columnas. Desarrollada por Avinash Lakshman y Prashant Malik en 2008, su código fuente está escrito en Java. En el año 2009 fue adoptada por el programa *Incubator* de *Apache*, convirtiéndose en un proyecto de alto nivel e introduciéndose como parte de los proyectos de la *Apache Foundation* en 2010 [8].

Figura 4. Logotipo de *Cassandra*

Las principales características de *Cassandra* son la escalabilidad lineal y la alta disponibilidad de los datos mediante la redundancia de estos en los distintos nodos, empleando distintas estrategias configurables. Para alcanzar este hecho, implementa comunicación asíncrona entre nodos, permitiendo que cualquiera de ellos pueda atender cualquier solicitud. Por esto, se promueve el despliegue del máximo número de nodos posibles, consiguiendo tolerancia a fallos.

Cassandra implementa su propio lenguaje de consulta, *Cassandra Query Language (CQL)*, como alternativa al tradicional *SQL*.

CAPÍTULO 3. ANÁLISIS

3.1. Functional Domain Driven Design

La aplicación de *DDD* está muy ligada a la programación orientada a objetos. En esta sección se busca conocer cómo transformar los elementos identificados en el capítulo anterior a las técnicas de la programación funcional, buscando la pureza que proporcionan los lenguajes de estas características. En primer lugar, es imprescindible estudiar un ejemplo de *DDD* desarrollado a través de la *OOP*.

Fragmento 4. Ejemplo de *DDD* a través de la *OOP*

```
// Elementos comunes
type Item = String
type Customer = String
// Agregado
class Cart(val customer: Customer, val creationDate: Date) {
  // Estado mutable
  var items: Map[Item, Int] = Map.empty
  // Operaciones sobre el estado mutable
  def removeItem(item: Item) = {
    if(!items.contains(item)) throw new Exception("Item not in cart")
    else items -= item
  }
  def addItem(item: Item, quantity: Int) =
    items += (item -> quantity)
}

val a = new Cart("User123", "today") // Items()
a.addItem("Product1", 10) // Items(Product1 -> 10)
a.addItem("Product2", 3)  // Items(Product1 -> 10, Product2 -> 3)
a.removeItem("Product1")  // Items(Product2 -> 3)
a.removeItem("Product1")  // Exception("Item not in cart")
```

El Fragmento 4 introduce la clase *Cart* como un agregado capaz de mantener el estado, realizar operaciones para modificarlo e incluso comprobar que se puedan realizar las operaciones. Al generar una instancia de este y realizar la misma operación varias veces, se puede observar cómo se actualizan los productos contenidos en múltiples ocasiones. Este fragmento de código posee dos características impuras, la primera es un estado mutable, no garantizando el *thread-safe*, mientras que la segunda es la excepción, tomando como salida de una función un valor inesperado.

Ahora que se han identificado los elementos de *DDD* y los problemas que afectan a este sencillo ejemplo escrito con orientación a objetos, es posible conseguir una primera versión funcional al eliminar la mutabilidad impuesta al estado.

Fragmento 5. Modificación del Fragmento 4 eliminando la mutabilidad del estado

```
// Elementos comunes
type Item = String
type Customer = String
// Agregado
class Cart(val customer: Customer, val creationDate: Date,
           val items: Map[Item, Int] = Map.empty) { // Estado inmutable

  def removeItem(item: Item) = { // Operaciones
    if(!items.contains(item)) throw new Exception("Item not in cart")
    else new Cart(customer, creationDate, items - item)
  }
  def addItem(item: Item, quantity: Int) =
    new Cart(customer, creationDate, items + (item -> quantity))
}
val a = new Cart("User123", "today")// Items()
val a1 = a.addItem("Product1", 10)  // Items(Product1 ->10)
val a2 = a.addItem("Product1", 10)  // a1.items == a2.items
val a12 = a1.addItem("Product2", 3) // Items(Product1 ->10,Product2 ->3)
```

Como se puede comprobar en el Fragmento 5, a través del estado inmutable, al aplicar la función *addItem* con mismo producto *Product1* y cantidad *10* sobre el mismo valor del carro *a* repetidas veces, siempre se obtiene el mismo resultado, con lo que se alcanza un primer acercamiento funcional.

¿Es posible abstraer este fragmento de un modo aún más funcional? La respuesta es afirmativa, existen dos modificaciones necesarias. Las funciones *addItem* y *removeItem* pueden abstraerse a un elemento nuevo del dominio, un servicio. Por otra parte, esta última función puede producir un efecto lateral si no se encuentra el producto en el carro, una excepción.

Fragmento 6. Ejemplo de *Functional DDD*

```
import scala.util.{ Try, Success, Failure }

case class Cart(customer: Customer, creationDate: Date,
                items: Map[Item, Int] = Map.empty)
trait CartService {
  def removeItem(c: Cart, item: Item): Try[Cart] = {
    if(!c.items.contains(item)) Failure(new Exception("Item not in
cart"))
    else Success(c.copy(items = c.items - item))
  }
  def addItem(c: Cart, item: Item, quantity: Int): Try[Cart] =
    Success(c.copy(items = c.items + (item -> quantity)))
}
object CartService extends CartService

import CartService._
val a = new Cart("User123", "today")     // Cart(User123,today,Items())
val a1 = addItem(a, "Product1", 10)
// Success(Cart(User123,today,Items(Product1 -> 10)))
val a2 = addItem(a1.get, "Product2", 3)
// Success(Cart(User123,today,Items(Product1 -> 10,Product2->3)))
```

En el Fragmento 6 se han introducido múltiples cambios, a continuación, se encuentra la explicación de esta trasformación:

- La clase *Cart* ahora es un modelo inmutable en un *ADT*, un tipo de dato composicional.
- Mediante la abstracción de servicio se consigue una separación entre estado y comportamiento. Si se diseñan otros tipos de carros, el servicio adquiere la capacidad de manejar estos elementos sin que ellos se encarguen de mantener la lógica de negocio.
- Las funciones *addItem* y *removeItem* se han purificado con la abstracción pura *Try*, que retorna *Failure* si se produce un error o *Success* con el valor deseado.

Con este fragmento de código se ha conseguido dar un pequeño paso hacia Functional Domain Driven Design, pero pueden surgir dudas como *por qué la palabra class tiene un case delante*, *qué es un trait* o *cuál es la diferencia entre un val de un var*. Las siguientes secciones analizan los elementos necesarios para el diseño de la librería que se explicará en el capítulo 3, el lenguaje *Scala*, los patrones funcionales más importantes como las mónadas, la librería *Cats* y, por último, el conjunto de herramientas *Akka*.

3.2. Elementos del lenguaje Scala

En el último fragmento de código aparecen algunas palabras reservadas que pueden causar confusión como *val* y *var*, o el modificador case antes de *class*. A continuación, se explican en detalle los elementos introducidos en los bloques anteriores y aquellos que serán de utilidad en próximos fragmentos.

3.2.1. Val vs var

Cuando se definen elementos en *Scala*, es importante decidir si se quiere que estos sean modificables o no, aquí entran en juego *val* (del inglés, *value*) y *var* (del inglés, *variable*). El primero se emplea para definir valores inmutables, es decir, no se pueden reasignar valor una vez se ha definido un valor. Para aquellos casos en donde es necesario disponer de un dato modificable, se dispone de *var* para declararlo con esta propiedad, ejemplo en Tabla 2.

Tabla 2. Diferencias entre *val* (izquierda) y *var* (derecha)

```
val a = 2
val b = 10
val c = a + b // c -> 12
a = 1 // Error: Reassignment to val
c = b // Error: Reassignment to val
```

```
var a1 = 2
var b1 = 10
var c1 = a + b // c1 -> 12
a1 = 1 // a1 -> 1
c1 = b1 // c1 -> 10
```

3.2.2. Trait

Los *traits* definen abstracciones que se utilizan para especificar comportamientos a través de propiedades y métodos [9]. En la terminología Java es lo que se conoce como *interface* y la principal diferencia entre ambos es que en los *trait* es posible realizar una implementación de los métodos definidos mientras que en Java esta característica se ha incluido en *Java 8*.

Fragmento 7. Ejemplo de definición de *trait*

```
trait MyEqual[A] {
  // A implementar por el cliente
  def equal(other: A): Boolean
  // Implementados
  def different(other: A): Boolean = !equal(other)
  val === : A => Boolean = equal
  val !== : A => Boolean = different
}
// Cliente del trait
case class AnInt(a: Int) extends MyEqual[AnInt] {
  override def equal(other: AnInt): Boolean =
    this.a == other.a
}
AnInt(1) === AnInt(2) // false
AnInt(2) === AnInt(2) // true
AnInt(2) !== AnInt(4) // true
```

3.2.3. Case class

Una *case class* representa un contenedor de datos, princpalmente inmutables, que depende únicamente de los argumentos de su constructor [10]. Se puede ver como una especialización del termino *class*, donde el compilador incluye, de forma automática, un construtor con sintaxis de inicialización compacta (no necesita la palabra reservada *new*), puede ser descompuesto a través de *pattern matching* y genera los métodos *equals* y *toString*, un ejemplo figura en Tabla 3. A su vez, estas presentan un pequeño inconveniente, una *case class A* no puede ser ancestro de otra *case class B*.

Tabla 3. Diferencias entre *class* (izquierda) y *case class* (derecha)

```
class IntClass(val data: Int)
val c1 = new IntClass(1)
val c2 = new IntClass(1)
c1 == c2 // false
println(c1) // IntClass@2b27cc70
```

```
case class IntCaseClass(data: Int)
val cc1 = IntCaseClass(1)
val cc2 = IntCaseClass(1)
cc1 == cc2 // true
println(cc1) // IntCaseClass(1)
```

3.2.4. Object y case object

Cuando es necesario definir una serie de métodos y valores que no se encuentran ligados a una instancia especifica de una clase, sino que pueden ser parte del ámbito global de la aplicación, estos deben incluirse en lo que se denomina *object*, un elemento que, al definirlo como tal, el compilador genera una única instancia (en inglés, *singleton*) accesible por el nombre del mismo [11]. En comparación con el lenguaje Java, un *object* se corresponde con los elementos que se definen en una clase o interfaz a través del modificador *static*.

Un *object* puede estar definido de forma independiente para, por ejemplo, unificar funcionalidades en módulos, o complementar una clase o *trait*. En este caso, se le denomina *companion object*, su nombre debe corresponderse con el elemento al que acompaña y entre ambos existe visibilidad de sus componentes. Los *objects* admiten también el modificador *case*, aportando información del mismo modo que lo realiza una *case class*.

Fragmento 8. Ejemplo de definición de una clase y su *companion object*

```
class Account private() // Constructor privado para la clase
object Account {
  var instancesCount = 0
  def createAccount = {
    val a = new Account // Puede acceder al constructor
    instancesCount += 1
    a
  }
}

val fail = new Account() // Error: constructor Account in class Account
                         //cannot be accessed
val a1 = Account.createAccount
val b2 = Account.createAccount
println(Account.instancesCount) // 2
```

3.3. La programación funcional en profundidad

Varias características de la programación han sido ya mencionadas, tales como las funciones de orden superior, la inmutabilidad de los datos o la transparencia referencial. Esta sección se centra en introducir algunas construcciones basadas en la programación funcional que son de utilidad, como los tipos de datos algebraicos, la genericidad o las construcciones puramente funcionales.

3.3.1. Tipos de datos algebraicos

Un *ADT* es un tipo definido por composición de otros. Esta se basa en la generación de diferentes constructores o variantes, y la información contenida en cada constructor o tuplas [12].

El Fragmento 9 muestra un ejemplo de definición de tipo de dato algebraico basado en listas de números.

Fragmento 9. Definición del *ADT* Lista de Enteros

```
sealed trait ListInteger
case class NodeListInteger(i: Int, next: ListInteger) extends ListInteger
case object NilListInteger extends ListInteger
```

Basado en la definición teórica de *ADT*, este ejemplo muestra los elementos descritos, explicados a continuación:

- El *trait ListInteger* define el tipo algebraico. Importante sellar la definición del *ADT* para que no pueda ser definida fuera del ámbito en el que está, facilitando la localización de los elementos.
- *NodeListInteger* y *NilInteger* constituyen los constructores del *ADT*. La definición de múltiples variantes se les denomina tipo suma, ya que el conjunto de valores representado por este *ADT* será la suma de estos.

- La definición de un constructor que requiera de información, tupla, se define a través de *case classes*, incluyendo métodos como *copy* o un extractor para el *pattern matching*. Distintos tipos suma en un mismo *ADT* no requieren que estas tuplas sean de la misma longitud. Este es *NodeListInteger*, que requiere una tupla donde aparece el valor que contiene un nodo y el elemento siguiente de la lista, que puede ser otro *NodeListInteger* o un tipo vacío, *NilListInteger*.
- La definición de un constructor que no necesite información extra se realiza a través de *case objects*, en el ejemplo está representado por *NilListInteger*.

Un punto importante de los *ADT* es que poseen una estructura composicional, está compuesto por sumas y productos. Además, una clase puede definirse por extensión. En contra, los *ADTs* no encapsulan comportamiento y deben ser definidos en otros elementos externos.

3.3.2. Genericidad

En la programación funcional es importante comprobar si se puede generalizar las funciones con el fin de modularizar y reutilizar los componentes el mayor número de veces posibles [12].

El Fragmento 10 muestra otro tipo basado en el *ADT* del Fragmento 9, esta vez con cadenas de caracteres.

Fragmento 10. Definición del *ADT* lista de cadenas de caracteres

```
sealed trait ListString
case class NodeListString(s: String, next: ListString) extends ListString
case object NilListString extends ListString
```

Se puede comprobar que la definición es la misma, salvo el tipo del valor que contiene un *NodeListX*. ¿Es posible unificar estas definiciones un tipo? La respuesta es afirmativa y se muestra en el Fragmento 11.

Fragmento 11. Definición generalizada del *ADT* lista

```
sealed trait List[+A]
case class Node[A](v: A, next: List) extends List[A]
case object Nil extends List[Nothing]
```

3.3.3. Manejo de errores con y sin excepciones

El manejo de errores a través del lenguaje *Scala* se hace a través de excepciones, pero, ¿no es una excepción un efecto inesperado o lateral? Un claro ejemplo es la función dividir dos números, que se muestra en el Fragmento 12.

Fragmento 12. Método dividir dos enteros, lanzando una excepción

```
class DivideByZeroException extends Exception
def divide(dividend: Int, divisor: Int): Float = {
  if(divisor == 0) throw new DivideByZeroException
  else dividend.toFloat / divisor
}
List(4, 2, 0, 3, 9).map(d => divide(10, d))
// 2.5
// 5
// DivideByZeroException
```

Se observa que la ejecución se detiene al lanzar la excepción, es decir, se produce un efecto lateral. Entonces, ¿cómo se puede continuar y dar de lado la excepción? Una posibilidad es el tipo *Option*, donde puede existir el valor retornando *Some(data)* o *None* en caso contrario.

Fragmento 13. Método dividir dos enteros, devolviendo *Option*

```
def divideWithOption(dividend: Int, divisor: Int): Option[Float] = {
  if(divisor == 0) None
  else Some(dividend.toFloat / divisor)
}
List(4, 2, 0, 3, 9).map(d => divideWithOption(10, d))
//  Some(2.5)
//  Some(5.0)
//  None
//  Some(3.3333333)
//  Some(1.1111112)
```

En el Fragmento 13 se comprueba que la ejecución es completa, pero surge un nuevo problema, ¿por qué hay un valor desconocido *None*? ¿Ha sucedido algo? Con *Option*, se pierde toda la información del error que ha surgido. En caso de que no importe el error, esta sería la forma de manejarlo, pero si queremos conocer, hay que recurrir al tipo *Either*. Este tipo puede producir solo uno de los dos posibles valores, *Left* o *Right*, cada uno con su tipo correspondiente. El primero es común usarlo para marcar flujos incorrectos mientras que el segundo se emplea para los resultados correctos.

Fragmento 14. Método dividir dos enteros, tratando la excepción con *Either*

```
def divideWithEither(dividend: Int, divisor: Int): Either[Exception,
Float] = {
  if(divisor == 0) Left(new DivideByZeroException)
  else Right(dividend.toFloat / divisor)
}
List(4, 2, 0, 3, 9).map(d => divideWithEither(10, d))
//  Right(2.5)
//  Right(5.0)
//  Left(DivideByZeroException)
//  Right(3.3333333)
//  Right(1.1111112)
```

El Fragmento 14 contempla como, aportando la información de la excepción, es posible completar el flujo y obtener ese efecto lateral, encapsulado para evitar tratar con él.

3.3.4. Funtores y Mónadas

En la programación funcional existen múltiples estructuras funcionales que nos ayudan a generalizar algunas aplicaciones importantes sobre tipos de datos. Dos de ellas son los Funtores y las Mónadas.

Un funtor es una función entre dos categorías que trasforma objetos de la primera a la segunda, de modo que se mantengan en la composición las identidades y los morfismos [12] [13]. Se corresponde con la definición de la función *map.*

Fragmento 15. Definición de funtor y ejemplos con los tipos *Option* y *List*

```
trait Functor[F[_]] {
  def map[A, B](as: F[A])(f: A => B): F[B]
}
val optionFunctor = new Functor[Option] {
  // Implementacion para Option
  def map[A, B](as: Option[A])(f: A => B): Option[B] = as match {
    case Some(a) => Option(f(a))
    case None => None
  }
}
val listFunctor = new Functor[List] {
  // Implementacion para List
  def map[A, B](as: List[A])(f: A => B): List[B] =
    as.foldLeft(List.empty[B])((acc, a) => acc :+ f(a))
}
optionFunctor.map(Option("Hello World!"))(str => str.length) // Some(12)
listFunctor.map(Lisest("Hello", "World", "!"))(_.length) // List(5, 5, 1)
```

El Fragmento 15 contiene la definición de funtor y dos posibles aplicaciones sobre el mismo, donde tenemos un contenedor de datos (*Option*) y una colección (*List*). Se puede comprobar a través de los ejemplos de aplicación la trasformación de *Option[String]* y *List[String]* a *Option[Int]* y *List[Int]*, a través de una función para calcular la longitud de las cadenas de caracteres, *length: String ⇒ Int*.

La Mónada se diseñó para especificar una secuencia de operaciones, representando programas similares a la programación estructurada [12][14]. Se entiende como una estructura para anidar funciones, manejando la composición de las mismas. La función principal a implementar es *flatMap*, y da funcionalidad a la *for-comprehension*.

En Fragmento 16 se muestra la definición de mónada y una aplicación sobre el tipo *Option*. También se puede ver en la definición que una mónada es un funtor, cuya implementación de la función *map* se puede conseguir únicamente basándose en las dos funciones principales de ésta.

Fragmento 16. Definición de mónada e implementación con el tipo *Option*

```
trait Monad[F[_]] extends Functor[F] {
  def returns[A](a: A): F[A]
  def flatMap[A, B](m: F[A])(f: A => F[B]): F[B]

  def map[A, B](as: F[A])(f: A => B): F[B] =
    flatMap(as)(a => returns(f(a)))
}
val optionMonad = new Monad[Option] {
  def returns[A](a: A): Option[A] = Some(a)
  def flatMap[A, B](m: Option[A])(f: A => Option[B]): Option[B] =
    m match {
      case Some(a) => f(a)
      case None => None
    }
}
import optionMonad._
flatMap(Some("Hello World")){str =>
  returns(if(str.contains("a")) true else false)
}
```

3.3.5. La Free Monad

La *Free Monad* es una implementación de la mónada con un constructor de tipos, con la que se consigue, sin necesidad de implementarla para el tipo que se indique, de ahí el termino *Free* [12][15]. Su principal uso es el diseño de *DSL* y, dadas las bondades de la mónada, se consiguen programas que pueden ser interpretados.

Fragmento 17. *ADT* de la *Free Monad*

```
sealed trait Free[F[_], A]
case class Return[F[_], A](a: A) extends Free[F, A]
case class Suspend[F[_], A](s: F[A]) extends Free[F, A]
case class FlatMap[F[_], A, B](s: Free[F,A],
                               f: A => Free[F, B]) extends Free[F, B]
```

A través del álgebra definido en Fragmento 17, la especificación del tipo *F* adecuadamente, se consigue la sintaxis del *DSL* que se desea representar, independientemente de la manera en que se ejecute el mismo, esto es, un lenguaje de efectos. En Fragmento 18 se muestra la definición de un *DSL* para tratar con entrada-salida.

Fragmento 18. *DSL* para entrada-salida

```
sealed abstract class IOEffect[T]
case object Read extends IOEffect[String]
case class Write(msg: String) extends IOEffect[Unit]
type IOProgram[T] = Free[IOEffect,T]
```

Se pueden dar múltiples implementaciones del mismo dependiendo de cuál sea el efecto que se quiere dar. En base al *DSL* del Fragmento 18, un lenguaje de entrada-salida puede interpretarse contra la consola o el sistema de ficheros de la máquina.

3.4. Cats, una librería especializada en la programación funcional

Cats implementa de forma genérica los tipos más importantes de la programación funcional, entre ellos, los explicados en el apartado anterior. Una *typeclass* es una construcción del sistema de tipos que soporta polimorfismo de modo que las funciones son polimórficas en los tipos que reciben. Las estructuras explicadas en el apartado anterior reciben esta denominación.

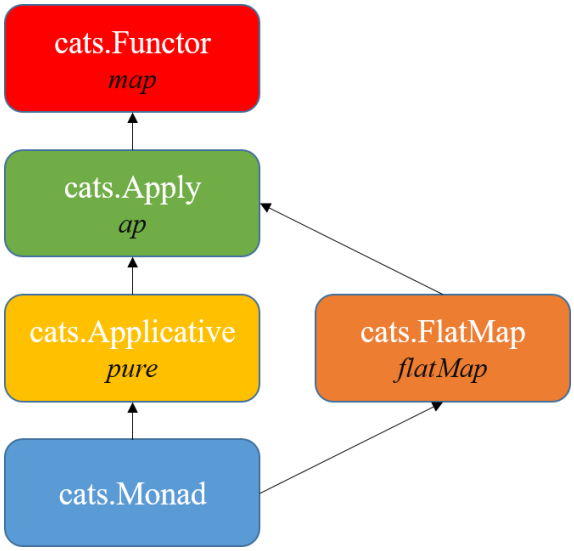

Figura 5. Organización de *TypeClasses* de la librería *Cats*

En la Figura 5 se observa la relación entre *typeclasses*, al igual que se explica en el apartado anterior la relación entre la mónada y el funtor.

Existen otros tipos, denominados tipos de datos, que actúan como contenedores. Los más importantes son:

- *Kleisli*, que permite la composición de funciones, es simplemente un envoltorio sobre la función *A => F[B]* [12][16].
- *Validated*, que actúa como acumulador de errores sobre un tipo a modo de validación [17].
- *Xor*, tipo isomorfo a *Either*, con la particularidad de que este posee una implementación dirigida por el tipo *Right*, es decir, las funciones de composición tales como *flatMap* o *map* actúan únicamente sobre el valor definido en *Right* y, si no está definido, se propaga el valor *Left* sin aplicar la función. Esto quiere decir que existe una implementación *Monad[Xor]* en donde se aplica la función únicamente sobre el valor *Right [18]*.

La mónada *Free* está implementada en un propio paquete independiente [15]. Está diseñada principalmente para detener la ejecución cuando sea necesario, ejecutando paso a paso a través de la función *step*, o ejecutar todo el programa usando *foldMap* o *compile*. Una particularidad de este desarrollo es que la interpretación de cualquier *Free* está basada en *Trampoline*, consiguiendo un tratamiento seguro de las llamadas a función al sustituirlas en la misma posición de la pila, del mismo modo que actúa una función con recursión de cola.

Free también posee características del lema de Yoneda, que dice que una Yoneda es isomorfo a *F[A]* para cualquier funtor *F*. La sintaxis `type IOProgram[T] = Free[IOEffect,T]` del Fragmento 18. *DSL* para entrada-salida se le denomina *Coyoneda Trick*, siendo este la dualidad de la Yoneda, y expresa que *Coyoneda f* es isomorfo a *f* de forma natural.

A través de la función *liftF* se posibilita crear constructores sencillos para los tipos de efectos definidos en los lenguajes.

Fragmento 19. *DSL* para entrada-salida con *Free* de *Cats* y un intérprete de consola

```
import cats.{ ~>, Id }
import cats.free.Free
import cats.free.Free.liftF
// ADT
sealed abstract class IOEffect[T]
case class Read() extends IOEffect[String]
case class Write(msg: String) extends IOEffect[Unit]
type IOProgram[T] = Free[IOEffect, T] // Coyoneda Trick
// Constructores
def read(): IOProgram[String] = liftF(Read())
def write(s: String): IOProgram[Unit] = liftF(Write(s))
// Interprete de consola
def consoleCompiler = new (IOEffect ~> Id) {
  import scala.io.StdIn.readLine
  def apply[A](fa: IOEffect[A]): Id[A] = fa match {
    case Read()      => readLine()
    case Write(str) => println(str)
  }
}
// Uso
val program: IOProgram[Unit] =
  for {
    _ <- write("Introduzca un saludo")
    s <- read() // "Hola Mundo!"
    _ <- write(s"Usted dijo: '$s'") // Usted dijo: 'Hola Mundo!'
  } yield ()
program.foldMap(consoleCompiler) // Ejecución del programa
```

En Fragmento 19 se puede ver la implementación de Fragmento 18 a través de la librería *Cats*. Aunque se defina el programa, este no se ejecuta hasta que es compilado, dando libertad de escoger un compilador diferente, por ejemplo, se puede decidir escribir y leer de un socket o sobre un fichero de texto. También es posible modificar la trasformación de *IOEffect* a Id por una transformación a futuros, con lo que se ejecutaría un programa asíncrono.

3.5. Elementos del conjunto de herramientas Akka

Akka comprende un amplio abanico de herramientas en la programación de actores, por lo que esta sección se divide en actores de *Akka*, *Akka Persistence* y *Akka Streams*, las tres partes más importantes del ecosistema.

3.5.1. Actores de Akka

Los actores son un principio de programación concurrente diseñada para evitar los bloqueos del sistema [19]. En *Akka*, un actor está definido a través del *trait Actor* y posee un método *receive*, cuya signatura es una función parcial de *Any* a *Unit*. Un actor debe implementarse para realizar una tarea en la que esté especializado y, si esta es un conjunto de varias, el sistema de actores, *ActorSystem* [20], provee mecanismos de gestión para ayudar al desarrollador a definir flujos entre múltiples actores, favoreciendo la modularización.

Fragmento 20. Definición de un actor que muestra el mensaje que se le envía

```
import akka.actor.{ Actor, ActorSystem, Props }
// Declaracion de implicitos necesarios por el entorno
implicit val system = ActorSystem("nombre-del-actorsystem")
implicit val executionContext = system.dispatcher
// Mensajes a enviar
case class Greeting(msg: String)
case object Greeted
// Actor
class Greeter(name: String) extends Actor {
  def receive = {
    case Greeting(msg) =>
      println(s"$name: $msg")
      sender ! Greeted
  }
}
// Ejecucion del actor
val greeter = system.actorOf(Props(new Greeter("Saludador1")))
greeter ! Greeting("Hello") // Saludador1: Hello
greeter ! Greeting("World") // Saludador1: World
greeter ! Greeting("Hello World Akka") // Saludador1: Hello World Akka
// Finalización del sistema de actores
system.terminate()
```

En el Fragmento 20 se encuentra la definición de un actor que se dedica a repetir el mensaje que se le ha enviado. Es recomendable encapsular los mensajes en algún tipo de elemento serializable, como una *case class*. El envío de mensajes se realiza a través del método *tell*, o su azúcar sintáctico *!*, aunque existen otros modos de envío como el patrón *ask*, donde se utiliza el símbolo *?*. La gran diferencia entre ambos patrones es que el primero es *fire-and-forget* mientras que en el segundo el emisor necesita recibir una confirmación por parte del receptor, basandose en futuros para este fin.

Los actores tienen un ciclo de vida definido a través de una serie de métodos que le permiten reiniciar el actor si este desaparece por algún problema, por ejemplo, una excepción. Algunos métodos del ciclo de vida más importantes son: *preStart*, *preRestart*, *postRestart*. Si se desea eliminar un actor del sistema, se recomienda enviarle un mensaje con una *PoisonPill*, incurriendo en una finalización abrupta del mismo.

El sistema de actores se encarga de coordinar la ejecución de los actores, gestionar su ciclo de vida y mantener la estructura jerárquica de los mismos. Por cada aplicación solo debe existir una instancia del *ActorSystem* ya que realiza la gestión de reserva de hilos del sistema, lo que podría suponer condiciones de carrera al coexistir múltiples instancias del mismo.

Con las nociones principales sobre cómo trabajar con actores, ahora se trata con otros modelos de actores con una serie de funcionalidades impuestas que nos aportan sencillez en la ejecución de las tareas.

3.5.2. Akka Persistence

Akka Persistence provee de actores capaces de persistir su estado interno y, a través de mecanismos de recuperación, alcanzar este cuando se reconstruya el actor [21]. La idea central de este tipo de actores es que los cambios que afectan al estado interno del actor sean persistidos y no el estado directamente, de modo que al volver a ejecutar estos cambios en orden se alcance este. *Akka Persistence* aporta comunicación punto a punto con entrega de mensajes al menos una vez.

Un *PersistentActor* es una especialización de *Actor* que implementa estas características, siendo requisito aportar el contenido a los métodos *receiveCommand* y *receiveRecover*, siendo el primero el que se ejecuta para modificar el estado interno del actor y persistir aquello que produzca cambios en el mismo, y el segundo el que se emplea para restaurar el estado a partir de la información persistida. Esta es la base del patrón *Event Sourcing*, cuya idea central es que el estado de una aplicación puede ser guardado a través de una secuencia de eventos, los cuales pueden ser leídos por otros subsistemas, siendo capaces de crear consultas específicas a los mismos.

En el Fragmento 21 se define un actor persistente basado en un contador. La sintaxis de los métodos *receiveCommand* y *receiveRecover* no están ligados a tipos, siendo funciones parciales de *Any* a *Unit*. El compilador no ayuda al usuario si se envía un mensaje al actor que este no sepa procesar, por lo que se definen dos elementos, *Command* y *Event*, que actuarán como envoltorios de la información. Los *ADTs* de Comandos y Eventos representan las acciones que el actor debe realizar sobre el estado y como informar de la acción que ha ejecutado.

Se observa un claro flujo en el actor que puede llegar a abstraerse si el usuario lo desea:

1. Recepción del comando e identificación del mismo.
2. Búsqueda del evento a producir.
3. Persistencia del evento.
4. Ejecución de una acción sobre el estado en base al evento que ha sido persistido.

Un elemento que se observa es la función *saveSnapshot*. Las *Snapshots* son imágenes del estado interno del actor creadas en un momento específico y que actuarán como estado base para la recuperación del estado. Se utilizan principalmente cuando ejecutar todos los eventos persistidos puede ser una tarea muy costosa.

Es importante reseñar que los actores persistentes no deben ser cerrados mediante *PoisonPill*, ya que, en ese caso, detendrá su acción inmediatamente, eliminando los comandos a procesar que tenga encolados. Para dicha acción es preferible que el mismo actor sepa cuando cerrarse, evitando encolar mensajes a partir del aviso.

Akka Persistence requiere de un destino donde persistir los datos, denominado *Journal*, donde existen múltiples opciones desarrolladas como *plugins*. Las más importantes son *Akka*

Persistence Cassandra, desarrollada por Martin Krasser y mantenida por *Akka* [22], y *Akka Persistence InMemory* de Dennis Vriend [23].

Fragmento 21. Implementación de un actor persistente

```
object PersistenceElements {
  trait Cmd // ADT Comandos
  case object Increment extends Cmd
  case object Decrement extends Cmd
  final case class MultiplyBy(n: Int) extends Cmd
  trait Evt // ADT Eventos
  case object Incremented extends Evt
  case object Decremented extends Evt
  final case class Multiplied(n: Int) extends Evt
  // Envoltorios
  final case class Command(c: Cmd)
  final case class Event(e: Evt)
  case object Ack
  case object Shutdown
}
import PersistenceElements._
// Definición del actor persistente
import akka.persistence.{ PersistentActor, SnapshotOffer }
class CounterPersistentActor extends PersistentActor {
  def persistenceId: String = "handler-1"
  var internalState: Long = 0
  def receiveCommand: Receive = {
    case Command(c) =>
      val e = c match {
        case Increment => Incremented
        case Decrement => Decremented
        case MultiplyBy(n) => Multiplied(n)
      }
      persist(Event(e)){ case Event(e) => update(e) }
      sender ! Ack
    case "makeSnapshot" =>
      saveSnapshot(internalState)
      sender ! Ack
    case "printState" => println(s"State: $internalState")
    case Shutdown => context.stop(self)
  }
  def receiveRecover: Receive = {
    case SnapshotOffer(_, state: Long) => internalState = state
    case Event(e) => update(e)
  }
  val update: PartialFunction[Evt, Unit] = {
    case Incremented => internalState += 1
    case Decremented => internalState -= 1
    case Multiplied(v) => internalState *= v
  }
}
// Ejemplo de uso
val counter = system.actorOf(Props(new CounterPersistentActor))
counter ! "printState" // State: 0
counter ! Command(Increment) // state -> 1
counter ! Command(Increment) // state -> 2
counter ! Command(MultiplyBy(10)) // state -> 20
counter ! Command(Decrement) // state -> 19
counter ! "printState" // State: 19
counter ! Shutdown
```

3.5.3. Akka Streams

Akka Streams es una librería para transmisión de flujos de información como pequeños bloques de datos [24]. Está implementada por encima de la capa de actores, quienes al recibir y emitir mensajes pueden transferirlos de forma eficiente y estable, y basada en el núcleo de *Reactive Streams*, donde se especifican algunos elementos como el *backpressure*, reduciendo la carga advirtiendo que es necesario reducir la velocidad de transferencia.

Existen tres elementos principales que dan lugar a una gran cantidad de conceptos. A continuación, se explica cada uno de ellos:

- *Source[Out, Mat1]*, fuente de datos del *stream*.
- *Sink[In, Mat3]*, destino del *stream*, suele implicar la materialización del mismo en forma de *Future*.
- *Flow[In, Out, Mat2]*, tuberías que enlazan los componentes de los s*treams*. Si un *Flow* se conecta con un *Source* se obtiene un nuevo *Source*, siendo posible añadir más elementos, del mismo modo funciona con *Sink*.

Un flujo compuesto por, como mínimo, la combinación de los dos primeros elementos, es un *Graph* y, si es posible ejecutarlo, se le denomina *RunnableGraph*. Los *streams* pueden materializar valores en cualquiera de estas piezas, consiguiendo valores intermedios de la computación.

3.5.4. Akka Persistence Query

Akka Persistence posee un mecanismo, ya deprecado, para acceder a los eventos persistidos a través del *trait PersistentView*. Se recomienda utilizar una librería nueva nombrada *Akka Persistence Query*.

Akka Persistence Query posee múltiples consultas que se pueden realizar sobre los datos persistidos por *Akka Persistence* [25]. Las conexiones se definen a través de los *plugins* anteriormente explicados.

Los *plugins* de consulta no están forzados a definir todas las consultas posibles. Para conocer las consultas, *Akka Persistence Query* define un conjunto de *traits* que extenderá el *plugin* si desea implementar. El más importante, que permite la conexión con el destino, es *ReadJournal*. Entre ellas destacan:

- *AllPersistenceIdsQuery*, que retorna todos los identificadores de persistencia existentes en el destino.
- *EventsByPersistenceId*, con el que se obtiene los eventos persistidos con el identificador dado.
- *EventsByTag*, que aporta los eventos basándose en la búsqueda por etiquetas definidas en los eventos.

Todas estas consultas retornan un *Source[EventEnvelope]*, donde se recogerán los eventos a través de *Akka Streams*. *EventEnvelope* es un envoltorio generado al persistir eventos.

CAPÍTULO 4. DISEÑO E IMPLEMENTACIÓN

4.1. CQRS

La librería desarrollada sigue el patrón conocido como *Command-Query Responsability Segregation* **¡Error! No se encuentra el origen de la referencia.**. *CQRS* aplica el principio d e la *Separación de Comandos y Consulta (CQS)* en dos subsistemas diferenciados. Aunque este principio, ideado por Bertrand Meyer, fue diseñado para la programación imperativa, posee bondades características de la programación funcional, afirma que un método debe ser un comando que realice una acción, o una consulta que retorna datos, pero no ambos a la vez, favoreciendo la trasparencia referencial y las funciones carentes de efectos laterales. *CQRS* separa este lema en dos subsistemas, un sistema de comandos y un sistema de consultas.

A continuación, se explica cada uno de los subsistemas de forma independiente, relacionándolos a través de uno de los elementos principales, el *log de eventos*, también denominado *journal*.

4.1.1. Subsistema de comandos

El subsistema de comandos contiene la relación entre las ordenes que se le dan al sistema, comandos, y las acciones que ya ha realizado el sistema a partir de dichas ordenes, eventos. Un comando produce, como mínimo, un evento, tras comprobar que se cumplen las restricciones de negocio. A este manejador se le denomina *CommandHandler*. En contra, es necesario especificar las acciones a realizar por el subsistema cuando un *CommandHandler* emite un evento, lo que se denomina *EventHandler*. Una característica importante es que los eventos son acumulativos, es decir, no se puede eliminar un evento ya persistido, pero, a cambio, se permite añadir otros que compensen la no destrucción de la historia.

Al aplicar los conceptos de *DDD*, un agregado debe recibir comandos, comprobar que se cumplen las restricciones de negocio conocidas como invariantes, emitir eventos si se puede realizar la acción correspondiente y actualizar el estado una vez se haya emitido el evento correspondiente. Aplicando los elementos estudiados tenemos:

- Para definir los comportamientos es posible crear un álgebra que especifique un *CommandHandler* y el *EventHandler* correspondiente. Este *ADT* determina el comportamiento del agregado.
- Un agregado es un elemento que se encarga de analizar comandos y gestionarlos. Dado que se busca abstraer los conceptos a la programación funcional, es necesario un álgebra que reaccione ante la llegada de comandos. Una implementación del compilador de este *ADT* puede darse a través de actores persistentes.

- Es posible definir un álgebra para el flujo de la aplicación, con lo que se obtendría una abstracción más sobre el comportamiento del agregado.

Estas tres álgebras son definidas a través de la mónada *Free*, con lo que se pueden obtener interpretes independientes del tipo destino, abierto a ser *Id* o *Future*, entre otras mónadas. Es lo que se denomina un *HKT*.

4.1.2. Subsistema de consultas

El subsistema de consultas se encarga de abrir conexiones al *journal* y recibir los eventos persistidos en él. Dado que los eventos pueden aparecer en cualquier momento ya que no se conoce cuando serán introducidos, las conexiones a la fuente de datos se dejan abiertas para su espera. Estos elementos capaces de realizar la lectura del *journal* se les denomina Proyecciones (en inglés, *Projections*).

Las proyecciones se encargan de interpretar los eventos de la forma que se le especifique, ampliando la lógica del subsistema de comandos. Una proyección debe estar ligada al tipo de los eventos emitidos por el agregado que va a consultar, con lo que se evita tener inconsistencias manejándolos al tener que ligar los tipos. Para leer los mismos eventos del *journal* se pueden definir tantas proyecciones como sean necesarias. Las distintas proyecciones no necesitan identificar los mismos eventos aun procediendo del mismo agregado, con lo que se obtienen distintas consultas al sistema.

Una Vista (en inglés, *View*), es la representación de los datos que se desean consultar. Para poder acceder a estas vistas, es necesario implementar un elemento que la conecte con el objeto capaz de comunicarse la fuente de datos, este se conoce como *ProjectionView*.

Las *Sagas* son un tipo especial de proyección que, a partir de eventos, es capaz de comunicarse con otros agregados enviándoles comandos, lo que produce actualizaciones en los receptores.

Para poder mantener la información de estos elementos y que sea consultable en cualquier momento de la historia, es importante darles un mecanismo donde hacerlos residir. Aquí nacen los Repositorios, que mantienen el patrón *repository* [27]. Este patrón provee de una estructura para mantener centralizada los datos, siendo una pieza fácilmente intercambiable. Para alcanzar esta facilidad, es posible diseñar un *ADT* capaz de mantener las acciones básicas sobre cualquier almacenamiento, pudiendo definir un intérprete para dicha álgebra según las necesidades del mismo.

4.2. Implementando CQRS

Basado en el diseño expuesto en el apartado anterior es sencillo implementar el patrón *CQRS* de un modo funcional a través de la mónada *Free*. Este apartado muestra el *DSL* que actúa como interfaz para el desarrollador.

4.2.1. DSL del flujo de comandos-eventos

En Fragmento 22 se dispone del *DSL* creado para especificar los flujos del agregado. El tipo *CommandHandler* es una función parcial cuya entrada es un comando de tipo *C* y produce una secuencia de eventos de tipo *E*. Del mismo modo, un *EventHandler* recibe eventos del mismo *E* que produce *CommandHandler* y retorna información que será utilizará para el control de invariantes.

Fragmento 22. *DSL* para definir un flujo de comandos-eventos

```
type CommandHandler = PartialFunction[C, Seq[E]]
type EventHandler[A] = PartialFunction[E, A]

def handler(ch: CommandHandler): Flow[Unit]
def waitFor[A](eh: EventHandler[A]): Flow[A]
```

4.2.2. DSL del agregado

Un agregado define tres acciones básicas, cargar el agregado, manejar un comando y descargar el agregado del sistema. Un agregado solo sabrá manejar un *ADT* de comandos, es importante definir bien el dominio para estos tipos. En Fragmento 23 se especifica su sintaxis.

Fragmento 23. *DSL* para definir un agregado

```
def loadAggregate(aggregateId: String): AggregateClient[Ref]
def handleCommand(command: C)(ref: Ref): AggregateClient[Ref]
def unloadAggregate(ref: Ref): AggregateClient[Unit]
```

4.2.3. DSL del repositorio

Un repositorio para *CQRS* se centra en los modelos de persistencia de pares claves-valor y contiene únicamente tres acciones fundamentales, guardar, obtener y actualizar. Los tipos *K* y *V* se corresponden con el tipo de la clave y el del valor respectivamente. En Fragmento 24 se muestra.

Fragmento 24. *DSL* para definir un repositorio

```
def put[K, V](k: K)(v: V): Repository[Unit]
def get[K, V](k: K): Repository[Option[V]]
def update[K, V](k: K)(f: V => V): Repository[Unit]
```

4.2.4. DSL de la proyección

En el Fragmento 25 se tiene la definición del *DSL* de la proyección. Ésta es más simple al solo tener que manejar eventos del tipo *E* sin retornar devolver nada, definido a través de una función parcial.

Fragmento 25. *DSL* para definir una proyección

```
type Ref
type EventHandler = PartialFunction[E, Unit]
def createProjection(eh: EventHandler): ProjectionClient[Ref]
```

CAPÍTULO 5. APLICACIÓN EN ENTORNOS DISTRIBUIDOS

5.1. Software en los entornos distribuidos

Los sistemas distribuidos son un conjunto de sistemas físicos conectados entre sí a través de red y que trabajan conjuntamente para alcanzar un objetivo [28]. El principal método de comunicación entre los procesos de estas máquinas es el paso de mensajes.

La programación de software como pequeñas piezas está en auge dados los entornos de computación en la nube. Estos fragmentos, denominados microservicios, son unidades computacionales de tareas muy específicas [29].

Un sistema software de ingestión de datos puede contener varias de estas piezas, primero uno para contener el *API*, después otro para procesar la entrada de datos y un último para las transformaciones pertinentes de dichos datos. Una unidad de software compleja como puede ser este servicio se ha conseguido descomponer en múltiples unidades sencillas con las que formalizar un producto software distribuido.

Gracias a esta arquitectura es posible aprovechar más recursos al poder aplicar escalabilidad horizontal de un mismo microservicio, con lo que pueden surgir dudas sobre cómo trabajan conjuntamente estos nodos del mismo microservicio sin que se puedan solapar entre ellos. En el próximo apartado se introduce el teorema *CAP* para solventar estas cuestiones.

5.2. Teorema CAP

El teorema CAP o teorema de Brewer explica que un sistema distribuido no puede garantizar simultáneamente tres requisitos, consistencia, disponibilidad y tolerancia al particionado [30]. Eric Brewer lo introdujo en el año 1999 como el principio *CAP* y lo presentó como una conjetura durante el Simposio de los Principios de Computación Distribuida. A lo largo del año 2002, Nancy Lynch y Seth Gilbert demostraron que dicho principio era cierto, convirtiéndose así en teorema [31].

A continuación, se explica en detalle en qué consisten cada una de estas características y por qué es importante su presencia en un sistema distribuido.

5.2.1. Consistencia

La consistencia se basa en que todos los nodos de un sistema distribuido son capaces de acceder a la misma información al mismo tiempo. Al realizar una petición al sistema, cualquier nodo perteneciente al mismo es capaz de resolverla ya que todos poseen acceso a la misma información de forma consistente.

5.2.2. Disponibilidad

La disponibilidad es un requisito no funcional de fiabilidad que se centra en que, para cada petición realizada al sistema, este debe emitir una respuesta, sin importar el resultado de la misma. Esta característica dota al cliente del sistema de la seguridad de que, como mínimo, la petición ha sido recibida, pero no avala que esta se haya procesado satisfactoriamente.

Es importante entender que este rasgo implica únicamente la confirmación de la petición y no el tiempo que se tarda en emitir y recibir la respuesta.

5.2.3. Tolerancia al particionado

La tolerancia al particionado se refiere a la capacidad de un sistema de continuar operando a pesar de la perdida de algún nodo, por ejemplo, por problemas de comunicación tales como la caída de la red.

5.3. Teorema CAP *(cont.)*

Una vez se entienden estas tres características de los sistemas distribuidos, es posible determinar las composiciones que se pueden garantizar simultáneamente en un sistema distribuido.

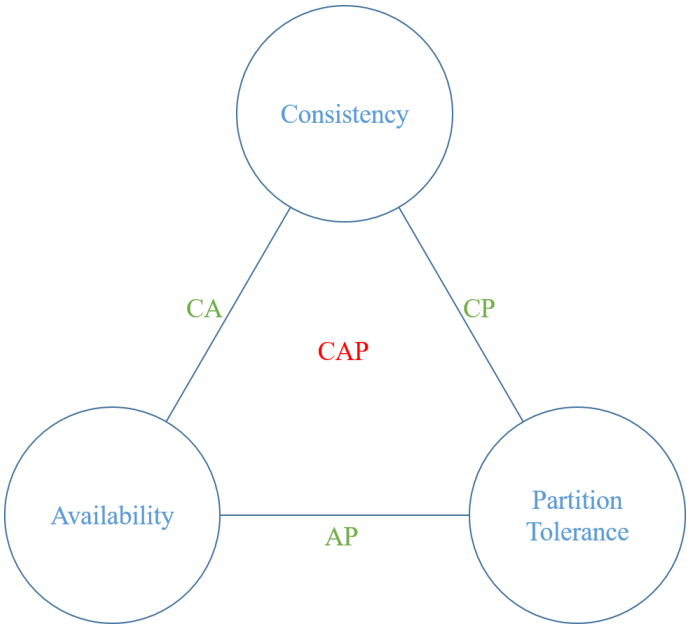

Figura 6. Teorema CAP

En la Figura 6 se puede comprobar que existen tres posibles combinaciones:

- Disponibilidad y Tolerancia al particionado (***AP***), en el que la información no será replicada a todos los nodos cuando el sistema se particiona, a costa de estar disponible para procesar la petición.
- Consistencia y Tolerancia al particionado (***CP***), donde para garantizar la consistencia de las distintas particiones, el sistema debe poder procesar la petición en todos sus nodos al mismo tiempo. Dada una partición del sistema, esta dejará de estar disponible y rechazará el procesamiento de peticiones hasta que se pueda deshacer la partición.
- Consistencia y Disponibilidad (***CA***), este caso contempla que cada operación será procesada y replicada al resto de los nodos. En este caso especial solo se puede dar en situaciones en que no haya particionado del sistema y, en caso de particionarse, el sistema debe fallar, no asegurando la disponibilidad, por lo que se convertiría en un caso de *CP*.

En el siguiente apartado se indicará cómo puede afectar este teorema al introducir los entornos distribuidos y *DDDD*.

5.4. Teorema CAP en entornos distribuidos y DDDD

Al aplicar el teorema *CAP* a la definición de los sistemas distribuidos se puede afirmar que la tolerancia al particionado es innegociable ya que pueden producirse fenómenos que realicen la partición de los nodos, por ejemplo, la caída de las comunicaciones entre estos.

Con la *P* fijada, existen dos posibilidades a la hora de escoger los requisitos que debe cumplir el sistema, consistencia o disponibilidad. Existen ventajas e inconvenientes que hay que tener en cuenta a la hora de decidir entre uno u otro.

5.4.1. ¿Qué es DDDD?

DDDD trata de resolver como actuar frente a la clusterización de los algoritmos implementados en *DDD*. Para actuar hay que tener en cuenta los hechos explicados sobre el teorema *CAP* y su aplicación sobre los sistemas distribuidos. Un elemento importante en este caso es el agregado raíz, encargado de distribuir la tarea entre los agregados.

5.4.2. El teorema CAP en DDDD

Si se desea que exista consistencia de datos (*CP*), no puede haber múltiples instancias de un tipo de agregado modificándose en el mismo agregado raíz, por lo que este no tendría disponibilidad para atender otros comandos hasta que finalice el procesamiento. A esto se le denomina *Strong Consistency*, conseguida a través de *Akka Persistence*.

En caso de preferir la disponibilidad del servicio (*AP*), surge el problema de que los datos no se encuentran replicados en todos los agregados del mismo tipo, por lo que un agregado podría actuar frente a un comando que aún no puede procesar al no disponer de toda la

información. Para solventar este modelo llamado *Eventual Consistency*, existe una técnica de sincronización del conocimiento de los agregados denominada Compensación. *Eventuate*, desarrollada por el equipo *RedBull Media House Technology* y Martin Krasser [32].

CQRS adquiere importancia al trabajar con *DDDD* ya que el elemento de sincronización para los actores es el *journal*. Para realizar la distribución de comandos entre los distintos nodos aparece el concepto de oficina, donde cada agregado ha de registrarse, actuando como agregado raíz. Centrándose en *Akka Persistence*, la consistencia de los datos se realiza en la llamada al método *persist* del *PersistentActor* [33].

CAPÍTULO 6. CONCLUSIONES Y TAREAS FUTURAS

6.1. Conclusiones

Este trabajo representa la investigación de metodologías y técnicas para el desarrollo de productos software ricos en el conocimiento del sector que se desea plasmar, en este caso, a través de *Domain Driven Design*, así como las fuertes diferencias entre programación imperativa y funcional, con énfasis sobre esta última. La aplicación de las técnicas descritas junto al conocimiento y la imaginación del desarrollador pueden generar programas escritos de forma sencilla y con semántica compleja.

Dado el gran abanico de capacidades de la programación funcional, aunque a primera vista complejas y sencillas tras interiorizar los conceptos, aportan al equipo de programadores seguridad sobre los algoritmos que implementan y elementos que pueden ser reutilizables por múltiples aplicaciones. La abstracción de Tipos de Datos Algebraicos a través de la *Free Monad* fomenta la creación de Lenguajes de Dominio Especifico y abre la puerta a los desarrolladores a reutilizar un mismo lenguaje aportando la manera de la que este se ha de comportar, a través de trasformaciones naturales.

Scala, el lenguaje empleado a lo largo de este trabajo, y las herramientas y librerías, Akka y Cats, aportan grandes útiles para los programadores, la mayoría de ellos dados por las propias comunidades de desarrolladores. Este hecho contribuye a la creación de instrumentos que pueden ser útiles a lo largo de todo el mundo, fomentando el software libre.

El diseño de una librería basada en el patrón *Command-Query Responsability Segregation* mediante lenguajes funcionales implica que los desarrolladores deben identificar y tratar con cuidado los elementos que forman el dominio, como el lenguaje, permitiendo a otros el fácil entendimiento de software robusto y fiable.

El auge de la computación en la nube y el software distribuido dan lugar a un tratamiento delicado acerca de las decisiones a tomar dependiendo de los requisitos del negocio. Los microservicios poseen un gran papel en este avance dada su simpleza y la potencia que proveen al disponer de arquitecturas capaces de escalar en función de las necesidades. Las características expresadas sobre *Distributed Domain Driven Design* aportan una base para la programación de este tipo de software.

6.2. Tareas futuras

La creación de una librería para facilitar a los programadores el desarrollo de sus aplicaciones forma una experiencia colaborativa y rica en conocimiento. La librería aquí descrita, empleando el patrón CQRS, supone un punto de partida para el diseño de software colaborativo y la ampliación de esta.

- Introducir en el lenguaje del agregado un mecanismo más formal para el control de invariantes, soporte para snapshots y ampliar su interprete para trabajar en entornos distribuidos.
- Revisar el lenguaje de dominio especifico que sirve de interfaz, convirtiéndolo en atractivo para los usuarios del mismo.
- Diseñar e implementar un *testkit* sobre la librería, lo que permite a los programadores probar los elementos del dominio, así como los flujos que la aplicación debe seguir en función del estado del agregado y los posibles comandos que recibe el sistema, mostrando estados inalcanzables. Este módulo de pruebas será ampliable a otros elementos como las proyecciones y vistas.

REFERENCIAS

[1] Evans, E. (2004). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.

[2] Ghosh, D. (Publication in September 2016). Functional and Reactive Domain Modeling. Manning Publications Co.

[3] Programación funcional https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional (Fecha de consulta: 04/06/2016).

[4] Funciones de orden superior, http://docs.scala-lang.org/tutorials/tour/higher-order-functions (Fecha de consulta: 05/06/2016).

[5] Odersky, M., Spoon, L., & Venners, B. (2008). Programming in scala. Artima Inc.

[6] Roestenburg, R., Bakker, R., & Williams, R. (Publication in August 2016). Akka in Action. Manning Publications Co.

[7] Librería Cats, http://typelevel.org/cats/index.html (Fecha de consulta: 07/06/2016).

[8] Apache Cassandra, http://cassandra.apache.org/ (Fecha de consulta: 07/06/2016).

[9] Traits, http://docs.scala-lang.org/tutorials/tour/traits (Fecha de consulta: 10/06/2016)

[10] Case Classes, http://docs.scala-lang.org/tutorials/tour/case-classes (Fecha de consulta: 10/06/2016).

[11] Singleton Objects, http://docs.scala-lang.org/tutorials/tour/singleton-objects (Fecha de consulta: 10/06/2016).

[12] Chiusano, P., & Bjarnason, R. (2014). Functional programming in Scala. Manning Publications Co.

[13] Functor en Cats, http://typelevel.org/cats/tut/functor.html (Fecha de consulta: 12/06/2016).

[14] Monad en Cats, http://typelevel.org/cats/tut/monad.html (Fecha de consulta: 12/06/2016).

[15] Free Monad en Cats, http://typelevel.org/cats/tut/freemonad.html (Fecha de consulta: 12/06/2016).

[16] Kleisli en Cats, http://typelevel.org/cats/tut/kleisli.html (Fecha de consulta: 14/06/2016).

[17] Validated en Cats, http://typelevel.org/cats/tut/validated.html (Fecha de consulta: 14/06/2016).

[18] Xor en Cats, http://typelevel.org/cats/tut/xor.html (Fecha de consulta: 14/06/2016).

[19] Actor Model, http://doc.akka.io/docs/akka/current/scala/actors.html (Fecha de consulta: 16/06/2016).

[20] Actor Systems, http://doc.akka.io/docs/akka/current/general/actor-systems.html (Fecha de consulta: 16/06/2016).

[21] Persistence, http://doc.akka.io/docs/akka/current/scala/persistence.html (Fecha de consulta: 17/06/2016).

[22] Github Akka Persistence Cassandra, https://github.com/akka/akka-persistence-cassandra (Fecha de consulta: 17/06/2016).

[23] Github Akka Persistence InMemory, https://github.com/dnvriend/akka-persistence-inmemory (Fecha de consulta: 17/06/2016).

[24] Streams Quickstart, http://doc.akka.io/docs/akka/current/scala/stream/stream-quickstart.html (Fecha de consulta: 18/06/2016).

[25] Persistence Query, http://doc.akka.io/docs/akka/current/scala/persistence-query.html (Fecha de consulta: 18/06/2016).

[26] Fowler, M. (2011). Cqrs. Command Query Responsibility Segregation), http://martinfowler (Fecha de consulta: 20/06/2016).

[27] Repositorio, https://msdn.microsoft.com/en-us/library/ff649690.aspx?f=255&MSPPError=-2147217396 (Fecha de consulta: 21/06/2016).

[28] Sommerville, I., & Galipienso, M. I. A. (2005). Ingeniería del software. Pearson Educación.

[29] Coulouris, G. F., Dollimore, J., & Kindberg, T. (2005). Distributed systems: concepts and design. pearson education.

[30] Teorema CAP, http://robertgreiner.com/2014/08/cap-theorem-revisited/ (Fecha de consulta: 23/06/2016).

[31] Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51-59.

[32] Eventuate, http://rbmhtechnology.github.io/eventuate/ (Fecha de consulta: 24/06/2016)

[33] Akka Persistence and Eventuate, http://www.slideshare.net/mrt1nz/akka-persistence-and-eventuate (Fecha de consulta: 24/06/2016)

GLOSARIO

API	Application Programming Interface
DSL	Domain Specific Language
SQL	Structured Query Language
DBMS	Database Management System
ADT	Algebraic Data Type
HKT	Higher Kinded Type
CAP	Consistency, Availability and Partition tolerance

ANEXOS

Anexo A. Ejemplo de uso de la librería CQRS

Este ejemplo trata de modelar los subsistemas de comandos y consulta de una aplicación de reserva de productos o ventas.

El primer aspecto a tratar es la definición del dominio en función de comandos y eventos. En Fragmento 26 y Fragmento 27 se definen, centrándose en la mínima información que debe contener cada uno de ellos.

Fragmento 26. Dominio de ventas

```
object salesDomain {
  type Identifier = String

  object commands {
    sealed trait SalesCmd {val reservationId: Identifier}

    final case class CreateReservation(
      reservationId: Identifier, shopId: Identifier, customer: Customer
    ) extends SalesCmd

    final case class ConfirmReservation(
      reservationId: Identifier
    ) extends SalesCmd

    final case class CancelReservation(
      reservationId: Identifier
    ) extends SalesCmd

    final case class CloseReservation(
      reservationId: Identifier
    ) extends SalesCmd

    final case class ReserveProduct(
      reservationId: Identifier, product: Product, quantity: Int
    ) extends SalesCmd

    final case class ChangeReservedProductQuantity(
      reservationId: Identifier, product: Product, quantity: Int
    ) extends SalesCmd

    final case class DeleteReservedProduct(
      reservationId: Identifier, product: Product
    ) extends SalesCmd

    final case class UpdatePaymentInfo(
      reservationId: Identifier, paymentInfo: PaymentInfo
    ) extends SalesCmd

    final case class UpdateShipmentInfo(
      reservationId: Identifier, shipmentInfo: ShipmentInfo
    ) extends SalesCmd
   }
```

Fragmento 27. Dominio de ventas (cont.)

```
  object events {
    sealed trait SalesEvt {val reservationId: Identifier}

    final case class ReservationCreated(
      reservationId: Identifier,
      shopId: Identifier,
      customer: Customer,
      date: DateTime
    ) extends SalesEvt

    final case class ReservationConfirmed(
      reservationId: Identifier,
      date: DateTime
    ) extends SalesEvt

    final case class ReservationCancelled(
      reservationId: Identifier,
      date: DateTime
    ) extends SalesEvt

    final case class ReservationClosed(
      reservationId: Identifier,
      date: DateTime
    ) extends SalesEvt

    final case class ProductReserved(
      reservationId: Identifier,
      product: Product,
      quantity: Int,
      date: DateTime
    ) extends SalesEvt

    final case class ReservedProductQuantityChanged(
      reservationId: Identifier,
      product: Product,
      quantity: Int,
      date: DateTime
    ) extends SalesEvt

    final case class ReservedProductDeleted(
      reservationId: Identifier,
      product: Product,
      date: DateTime
    ) extends SalesEvt

    final case class PaymentInfoUpdated(
      reservationId: Identifier,
      paymentInfo: PaymentInfo,
      date: DateTime
    ) extends SalesEvt

    final case class ShipmentInfoUpdated(
      reservationId: Identifier,
      shipmentInfo: ShipmentInfo,
      date: DateTime
    ) extends SalesEvt
  }
}
```

Tras comprobar la definición de los *ADTs* de comandos y eventos para este dominio, es importante reseñar las entidades y *value objects* que forman el mismo en el Fragmento 28.

Fragmento 28. Entidades y *value objects* del dominio de ventas

```
object entities {
  object ReservationStatus extends Enumeration {
    type ReservationStatus = Value
    val Opened, Confirmed, Cancelled, Closed = Value
  }
  case class Reservation(
    reservationId: Identifier,
    status: ReservationStatus,
    shipmentInfo: Option[ShipmentInfo],
    paymentInfo: Option[PaymentInfo],
    orderProducts: Vector[OrderProduct] = Vector.empty
  )

  case class Customer(
    customerId: Identifier,
    fullName: String
  )

  case class OrderProduct(
    product: Product,
    quantity: Int
  )

  case class Product(
    shopId: Identifier,
    productId: Identifier,
    code: String,
    name: String,
    description: String,
    productType: String,
    price: Money
  )
}

object valueObjects {
  case class PaymentInfo(
    cardNumber: String,
    authorizedName: String,
    expirationMonth: String,
    expirationYear: String,
    securityNumber: String
  )

  case class ShipmentInfo(
    name: String,
    surname: String,
    address: String,
    zipCode: String,
    village: String,
    region: String
  )

  sealed trait Currency
  case object EUR extends Currency
  case class Money(amount: BigDecimal, currencyCode: Currency)
}
```

El punto de partida es el subsistema de comandos. Con los elementos del domino descritos, es el momento de mostrar el flujo que posee el sistema, aparecen en el Fragmento 29. En los comandos, manejados en la especificación de *handle*, se puede comprobar ciertos chequeos de invariantes a través de cláusulas *if*, por ejemplo, solo es posible confirmar una reserva si su estado es abierto y tiene productos incluidos o solo se pueden eliminar productos si estos se encuentran en la reserva.

Fragmento 29. Flujo de negocio en base a los comandos y eventos del domino

```
val startPointLogic: Flow[Unit] =
  handler {
    case CreateReservation(rid, sid, c) =>
      List(ReservationCreated(rid, sid, c, dateTimeNowUTC))
  } >>
  waitFor {
    case ReservationCreated(rid, sid, c, dt) =>
      Reservation(rid, Opened, None, None)
  } >>=
  fullLogic

def fullLogic(r: Reservation): Flow[Unit] =
  handler {
    case ConfirmReservation(rid) if r.status == Opened
      && r.orderProducts.nonEmpty && r.paymentInfo.nonEmpty
      && r.shipmentInfo.nonEmpty =>
      List(ReservationConfirmed(rid, dateTimeNowUTC))
    case CancelReservation(rid) =>
      List(ReservationCancelled(rid, dateTimeNowUTC))
    case CloseReservation(rid) =>
      List(ReservationClosed(rid, dateTimeNowUTC))
    case ReserveProduct(rid, p, q) if r.status == Opened
      && !r.orderProducts.exists(_.product == p) =>
      List(ProductReserved(rid, p, q, dateTimeNowUTC))
    case ChangeReservedProductQuantity(rid, p, q) if r.status == Opened
      && r.orderProducts.exists(_.product == p) =>
      List(ReservedProductQuantityChanged(rid, p, q, dateTimeNowUTC))
    case DeleteReservedProduct(rid, p) if r.status == Opened
      && r.orderProducts.exists(_.product == p) =>
      List(ReservedProductDeleted(rid, p, dateTimeNowUTC))
  } >>
  waitFor {
    case ReservationConfirmed(_, _) =>
      r.copy(status = Confirmed)
    case ReservationCancelled(_, _) =>
      r.copy(status = Cancelled)
    case ReservationClosed(_, _) =>
      r.copy(status = Closed)
    case ProductReserved(_, p, q, _) =>
      r.copy(orderProducts = r.orderProducts :+ OrderProduct(p, q))
    case ReservedProductQuantityChanged(_, p, q, _) =>
      val oldElem = r.orderProducts.find(_.product == p).get
      r.copy(orderProducts = r.orderProducts.filterNot(_ == oldElem) :+
        OrderProduct(p, q))
    case ReservedProductDeleted(_, p, _) =>
      val elem = r.orderProducts.find(_.product == p).get
      r.copy(orderProducts =
        r.orderProducts.filterNot(op => op.product == p))
  } >>=
  fullLogic
```

En Fragmento 30 se introduce un programa formado por la emisión de comandos al agregado. El operador >>= es azúcar sintáctico de la función *flatMap*, permitiendo la composición del programa. En este, se crea una reserva, añaden dos productos con una cantidad determinada, que será modificada repetidas veces antes de eliminar el primer producto y confirmar la reserva.

Fragmento 30. Ejemplo de emisión de múltiples comandos al agregado

```
val program: client.AggregateClient[Unit] =
  loadAggregate(aggregateId) >>=
  handleCommand(CreateReservation(reservationId, shopId, customer)) >>=
  handleCommand(ReserveProduct(reservationId, product1, quantity1)) >>=
  handleCommand(ReserveProduct(reservationId, product2, quantity1)) >>=
  handleCommand(ChangeReservedProductQuantity(reservationId, product1,
    quantity1)) >>=
  handleCommand(ChangeReservedProductQuantity(reservationId, product1,
    quantity2)) >>=
  handleCommand(ChangeReservedProductQuantity(reservationId, product2,
    quantity2)) >>=
  handleCommand(DeleteProduct(reservationId, product1)) >>=
  handleCommand(ConfirmReservation(reservationId)) >>=
  unloadAggregate

program.runNow
```

Tras finalizar con el subsistema de comandos hay que ver el subsistema de consultas.

En primer lugar, hay que definir las vistas y como actuaran frente a la llegada de eventos. Para este ejemplo se muestra únicamente una vista que actúa en relación a la gestión de productos, mostrado en Fragmento 31 y Fragmento 32.

Tras conocer la vista, hay que definir el repositorio y la fuente de datos, así como la proyección, que se encarga de trabajar con estos elementos, formando una imagen clara. El Fragmento 33 contiene la integración de estos elementos. Como repositorio se facilita una implementación en memoria y la fuente de los eventos persistidos por el agregado es la base de datos *Cassandra*.

Finalmente, solo hay que aludir al repositorio para obtener las vistas guardadas en el repositorio y mostrarlo al usuario, como en el Fragmento 34.

Fragmento 31. Vista y proyección sobre productos

```
case class OrderLineItemView(
  viewId: String,
  reservationItem: Option[OrderProduct]
) extends SalesBaseView

def orderLineItemsByReservationHandle(implicit rc: RepositoryCompiler)
: PartialFunction[SalesEvt, Unit] = {
  case ProductReserved(reservationId, product, quantity, _) =>
    Repository.put(reservationId + "-orderLineItemsByReservation") {
      OrderLineItemView(
        reservationId, Option(OrderProduct(product, quantity)))
    }.runNow
```

Fragmento 32. Vista y proyección sobre productos (cont.)

```
  case ReservedProductQuantityChanged(
    reservationId, product, quantity, dateCreated) =>
    Repository.update(reservationId + "-orderLineItemsByReservation") {
      v: OrderLineItemView => v.copy(
        reservationItem = v.reservationItem.map(item =>
          OrderProduct(product, quantity)))
    }.runNow
  case ReservedProductDeleted(reservationId, product, dateCreated) =>
    Repository.update(reservationId + "-orderLineItemsByReservation") {
      v: OrderLineItemView =>
        v.copy(reservationItem = None)
    }.runNow
  case _ => ()
}
```

Fragmento 33. Creación de una proyección con el repositorio y la fuente de datos

```
// define repo
implicit val repositoryCompiler = InMemoryRepository.repositoryCompiler

// define the data source
implicit val sources =
  CassandraEventStreamProvider.fromJournal(aggregateId, 0L,
Long.MaxValue).runNow

// create the projection actor
val projectionClient = ProjectionActorAlgrebra.build[SalesEvt]
import projectionClient._

// create the runnable projection
createProjection(orderLineItemsByReservationHandle).runNow
```

Fragmento 34. Acceso al repositorio para obtener las vistas

```
val orderLineItemsView = Repository.get[String, SalesBaseView](
  reservationId + "-orderLineItemsByReservation"
).runNow
val reservationsView = Repository.get[String,
SalesBaseView](reservationId + "-reservations").runNow
println(
  s"""Show from repo:
      |${orderLineItemsView.getOrElse("Not found a order line items by
reservation view")}
      |${reservationsView.getOrElse("Not found a reservations view")}
   """.stripMargin
)
```